Type Annotation (1A)

Young Won Lim
8/20/23

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using OpenOffice.

Type annotation (1)

In the programming language such as C or C++,
the data type must be declared before using a variable

declare an integer in C.

```
int a;
```

then we know that the variable "**a**" is of type integer.
then we would assign an integer to a.

```
a = 3;
```

the data type is not declared explicitly in Python

```
a = 3
print(type(a))
```

```
<class 'int'>
```

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (2)

As **a** is assigned an integer,
it belongs to the integer class itself,
without us having to say beforehand

```
a = int()
```

the type of **a** would change accordingly,
when it is assigned values from *other* data types.

```
a = 'hello'
print(type(a))
```

```
<class 'str'>
```

```
a = 3.14
print(type(a))
```

```
<class 'float'>
```

If Python is capable of determining the types itself,
why are even type annotations useful?

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (3)

Why Type Annotations

- identify where the type errors stem from.
- By using an IDE, the type annotations would allow you to access the built-in functions easier.
- When a variable is of no type, you cannot automatically access the built-in functions
- syntax-highlighting as a warning before you even run your code.
- more readable & understandable code.

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (4)

How Type Annotations

Example 1

the **ord()** function takes a string as input
and converts it into an integer by its the ASCII values

```
def my_function(a:int,b:str)->int:
    return a + ord(b)

print(my_function(3,'a'))
```

100

‘a’ = 97 (0x61)

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (5)

my_function() works perfectly,
since **b** has to be of type string and
‘a’ is of type string.

print(my_function(3,5))

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_19492/488431691.py in <module>
----> 1 print(my_function(3,5))

/tmp/ipykernel_19492/2046822703.py in my_function(a, b)
      1 def my_function(a:int,b:str)->int:
----> 2     return a + ord(b)

TypeError: ord() expected string of length 1, but int found
```

Here we receive a TypeError because we wrote an integer,
and not a string for **b**. Since **ord()** needs a string to function,
the code doesn't work.

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (6)

```
print(my_function('a','a'))
```

```
def my_function(a:int,b:str)->int:
    return a + ord(b)
```

OUTPUT:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_19492/1936946746.py in <module>
----> 1 print(my_function('a','a'))

/tmp/ipykernel_19492/2046822703.py in my_function(a, b)
      1 def my_function(a:int,b:str)->int:
----> 2     return a + ord(b)

TypeError: can only concatenate str (not "int") to str
```

here we receive another TypeError
because we wrote a string '**a**', and <u>not</u> an integer for **a**.

qhile **b** (**'a'**) is sucessfully <u>converted</u> into an integer by the **ord()** function,
it is yet <u>not</u> possible to <u>concatenate</u> strings and integers.

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (7)

```python
print(my_function(4.2,'a'))
```

101.2

```python
def my_function(a:int,b:str)->int:
    return a + ord(b)
```

'a' = 97 (0x61)

Interestingly our function *runs* now,
yet our argument **a** is of type float
and not integer as we declared at the very beginning.

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (7)

Example 2:

how the functions can be given variables
that can work without raising any errors.
but still be very *problematic*.

```
def add_together(a:int,b:str)->int:
    return a + b

def last_digit(a:int)->int:
    return a % 10

our_sum = add_together(38,57)
print(last_digit(our_sum))
```

```
5
```

```
def my_function(a:int,b:str)->int:
    return a + ord(b)
```

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (8)

```
our_sum = add_together(38,57)
print(last_digit(our_sum))


---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_19492/576833914.py in <module>
----> 1 print(return_the_last_digit(our_sum))

/tmp/ipykernel_19492/1809188113.py in return_the_last_digit(a)
      1 def return_the_last_digit(a:int)->int:
----> 2     return a % 10

TypeError: not all arguments converted during string formatting
```

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (9)

How can we run a TypeError check
before we actually run the program?

By installing **mypy** and running it
before you run your code,
you could avoid type errors.

Mypy checks your annotations and
ives a warning if a function is initialized
with the wrong datatype.

Mypy

All in all, type annotations are very useful and
it can save a lot of time for you and
it can make your code readable
or both yourself and the others.

https://python-course.eu/python-tutorial/type-annotations.php

Type annotation (1-1)

This *vague styling structure* comes partially
from Python being a dynamic typed language,
meaning that types are associated
with the variable's value at a point in time,
not the variable itself.

This language attribute means
that variables
can take on any value at any point and
are only type checked
when an attribute or method is accessed.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (1-2)

Consider the following code. In Python, this is *acceptable*.

```
age = 21
print(age)              # 21
age = 'Twenty One'
print(age)              # Twenty One
```

In the code above, the value of age is first an int (integer), but then we change it to a str (string) later on.

Every variable can represent any value at any point in the program.

That is the power of dynamic typing!

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (2)

Let's do the same thing in a statically typed language, like Java.

```
int age = 21;
System.out.print(age);
age = "Twenty One";
System.out.print(age);
```

We end up with the following error
because we are trying to assign "Twenty One" (a String)
to the variable age that was declared as an int.

Error: incompatible types: String cannot be converted to int

To work in a statically typed language,
we would have to use two separate variables and
use some assistive type-conversion method,
such as the standard **toString**() method.

```
int ageNum = 21;
System.out.print(ageNum);
String ageStr = ageNum.toString();
System.out.print(ageStr);
```

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (3)

This conversion works, but I really like the flexibility of Python,
and I don't want to sacrifice its *positive attributes*
as a dynamic, readable, and beginner-friendly language
just because types are difficult to *reason* about in most cases.

With this said, I also enjoy the readability of statically typed languages
for other programmers to know what type a specific variable should be!

So, to get the best of both worlds,
Python 3.5 introduced type annotations.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (4-1)

What Are Type Annotations?

Type Annotating is a new feature added in *PEP 484*
that allows adding type hints to variables.

They are used to inform someone reading the code
what the type of a variable should be expected.

This hinting brings a *sense* of statically typed control
to the dynamically typed Python.

This is accomplished by adding a given type declaration
after initializing/declaring a variable or method.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (4-2)

Why & How to Use Type Annotations

A helpful feature of statically typed languages is
that the value of a variable can always be known
within a specific domain.

For instance, we know string variables can only be strings,
ints can only be ints, and so on.

With dynamically typed languages,
its basically anyone's guess
as to what the value of a variable is or should be.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (5-1)

Annotating Variables

When annotating variables, it can be defined in the form

my_var: <type> = <value>

to create a variable named **my_var**
of the *given* **type** with the *given* **value**.

adds the **: int** when we declare the variable
to show that the variable **age** should be of type **int**.

```
age: int = 5
print(age)

# 5
```

I

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (5-2)

It is important to note that type annotations
do not affect the program's runtime in any way.

These hints are ignored by the interpreter and
are *solely* used to increase the readability
for other programmers and yourself.

But again, these type hints are not enforced are runtime,
so it is still up to the caller method / function / block
to ensure proper types are used.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (6-1)

Annotating Functions & Methods

We can use the *expected* variable's type
when writing and calling functions
to ensure we are passing and using parameters correctly.

If we pass a **str** when the function expects an **int**,
then it most likely will not work in the way we expected.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (6-2)

Consider the following code below:

```
def mystery_combine(a, b, times):
    return (a + b) * times
```

We know what that function is doing,

we do not know what **a**, **b**, or **times** are supposed to be?

we can *call* the **mystery_combine**
with *different* types of arguments.

```
print(mystery_combine(2, 3, 4))
# 20

print(mystery_combine('Hello ', 'World! ', 4))
# Hello World! Hello World! Hello World! Hello World!
```

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (7)

```
# Our original function
def mystery_combine(a, b, times):
    return (a + b) * times

print(mystery_combine(2, 3, 4))
# 20

print(mystery_combine('Hello ', 'World! ', 4))
# Hello World! Hello World! Hello World! Hello World!
```

what we pass the function,
two totally *different* results.

With integers we get some nice PEMDAS math,
but when we pass strings to the function,
we can see that the first two arguments are concatenated,
and that resulting string is multiplied times times.

Using type annotations, we can clear up
the purpose of this code

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (8-1)

```
def mystery_combine(a, b, times):
    return (a + b) * times

def mystery_combine(a:str, b:str, times:int)->str:
    return (a + b) * times
```

We have added **: str**, **: str**, and **: int**
to the function's parameters
to show what types they should be.

make clearer to read, reveal the purpose

We also added the **-> str** to show
that this function will return a **str**.

Using **-> <type>**, we can more easily show
the return value types of any function or method,
to avoid confusion by future developers!

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (8-2)

```
def mystery_combine(a, b, times):
    return (a + b) * times

def mystery_combine(a:str, b:str, times:int)->str:
    return (a + b) * times
```

Again, we can still call our code in the first, incorrect way,
but hopefully with a good review,
a programmer will see that they are using the function
in a way it was not intended.

```
print(mystery_combine(2, 3, 4))
# 20

print(mystery_combine('Hello ', 'World! ', 4))
# Hello World! Hello World! Hello World! Hello World!
```

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (8-3)

Type annotations and hints are incredibly useful
for teams and multi-developer Python applications.
It removes most of the guesswork from reading code!

We can extend this one step further
to handle default argument values.

We have adapted **mystery_combine** below
to use 2 as the default argument value of the **times** parameter.

This default value gets placed after the type hint.

```
def mystery_combine(a, b, times):
    return (a + b) * times
```

```
def mystery_combine(a:str, b:str, times:int)->str:
    return (a + b) * times
```

```
def mystery_combine(a:str, b:str, times:int = 2)->str:
    return (a + b) * times
```

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (9)

Type Hints with Methods

Type hints work very similarly with methods,
although it's pretty common
to leave off the type hint for self,
since that is implied to be
an instance of the containing class itself.

```
class WordBuilder:

    suffix = 'World'

    def mystery_combine(self, a: str, times: int) -> str:
        return (a, self.suffix) * times
```

very similar to the previous function-based example,
except we have dropped the **b** parameter
for a **suffix** attribute that is on the **WordBuilder class**.
Note that we don't need to explicitly add **: str** to the **suffix** definition
because most code editors will look at the default value for the expected type.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (10)

Available Types

The previous section handles many basic use cases of type annotations, but nothing is ever just basic, so let's break down some more complex cases and show the common types.
Basic Types

The most basic way to annotate objects is with the class types themselves. You can provide anything that satisfies a type in Python.

```
# Built-in class examples
an_int: int = 3
a_float: float = 1.23
a_str: str = 'Hello'
a_bool: bool = False
a_list: list = [1, 2, 3]
a_set: set = set([1, 2, 3])  # or {1, 2, 3}
a_dict: dict = {'a': 1, 'b': 2}

# Works with defined classes as well
class SomeClass:
    pass

instance: SomeClass = SomeClass()
```

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (11)

Complex Types

Use the typing module for anything more than a primitive in Python. It describes types to hint any variable of any type more detailed. It comes preloaded with type annotations such as Dict, Tuple, List, Set, and more! In the example above, we have a list-hinted variable, but nothing defines what should be in that list. The typing containers provided by the typing module allow us to specify the desired types more correctly.

Then you can expand your type hints into use cases like the example below.

```
from typing import Sequence

def print_names(names: Sequence[str]) -> None:
    for student in names:
        print(student)
```

This will tell the reader that names should be a Sequence of strs, such as a list, set, or tuple of strings.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (12)

Dictionaries work in a similar fashion.

```
from typing import Dict

def print_name_and_grade(grades: Dict[str, float]) -> None:
    for student, grade in grades.items():
        print(student, grade)
```

The Dict[str, float] type hint tells us that grades should be a dictionary where the keys are strings and the values are floats.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (13)

Type Aliases

If you want to work with custom type names, you can use type aliases. For example, let's say you are working with a group of \[x, y\] points as Tuples, then we could use an alias to map the Tuple type to a Point type.

```
from typing import List, Tuple


# Declare a point type annotation using a tuple of ints of [x, y]
Point = Tuple[int, int]


# Create a function designed to take in a list of Points
def print_points(points: List[Point]):
    for point in points:
        print("X:", point[0], "  Y:", point[1])
```

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (14)

Multiple Return Values

If your function returns multiple values as a tuple, wrap the expected output as a typing.Tuple[<type 1>, <type 2>, ...]

```
from typing import Tuple

def get_api_response() -> Tuple[int, int]:
    successes, errors = ... # Some API call
    return successes, errors
```

The code above returns a tuple of the number of successes and errors from the API call, where both values are integers. By using Tuple[int, int], we are indicating to a developer reading this that the function does return multiple int values.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (15)

Multiple Possible Return Types

If your function has a value that can take on a different number of forms, you can use the typing.Optional or typing.Union types.

Use Optional when the value will be be either of the given type or None, exclusively.

```
from typing import Optional

def try_to_print(some_num: Optional[int]):
    if some_num:
        print(some_num)
    else:
        print('Value was None!')
```

The above code indicates that some_num can either be of type int or None.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (16)

Use Union when the value can take on more specific types.

```
from typing import Union

def print_grade(grade: Union[int, str]):
    if isinstance(grade, str):
        print(grade + ' percent')
    else:
        print(str(grade) + '%')
```

The above code indicates that grade can either be of type int or str. This is helpful in our example of printing grades so that we can print either 98% or Ninety Eight Percent, with no unexpected consequences.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (17)

Working with Dataclasses

Dataclasses are a convenience class that provide automatically generated __init__ and __repr__ methods to an appropriate class. It reduces the amount of boilerplate code needed to create new classes that take in multiple keyword arguments to their constructor. These dataclasses use type hints and class-level attribute definitions to determine what keyword arguments and associated values can be passed to __init__ and printed by __repr__.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (18)

The following code is directly from the dataclasses documentation. It defines an InventoryItem that has three attributes defined on it, all using type hints; a name, unit_price, and quantity_on_hand .

```
from dataclasses import dataclass

@dataclass
class InventoryItem:
    """Class for keeping track of an item in inventory."""
    name: str
    unit_price: float
    quantity_on_hand: int = 0

    def total_cost(self) -> float:
        return self.unit_price * self.quantity_on_hand
```

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

Type annotation (19)

Using the type hints and @dataclass decorator, new InventoryItems can be created with the following code, and the dataclass will take care of mapping the keyword arguments to attributes.

```
common_item = InventoryItem(name='My Item', unit_price=2.99, quantity_on_hand=60)
other_item = InventoryItem(name='My Item', unit_price=2.99)  # uses default value of 10 quantity
```

An important note to @dataclasses is that any class attribute defined with a default value must be declared after any attributes without a default value. This means quantity_on_hand has to be declared after name and unit_price. This can get interesting when working with dataclasses that extend from a parent dataclass, so be careful, but the Python interpreter should catch these issues for you.

https://dev.to/dan_starner/using-pythons-type-annotations-4cfe

References

[1] Essential C, Nick Parlante
[2] Efficient C Programming, Mark A. Weiss
[3] C A Reference Manual, Samuel P. Harbison & Guy L. Steele Jr.
[4] C Language Express, I. K. Chun